CURS 6
Funcţii

O *funcţie* este un set de instrucţiuni, grupate sub un nume unic, care efectuează o prelucrare specifică a unor date în momentul în care este apelată.

Funcţiile sunt utilizate pentru a modulariza codul dintr-un program, prin împărţirea sa în subprograme (i.e., funcţii) care execută fiecare o singură prelucrare bine definită. În plus, funcţiile permit o reutilizare simplă şi sigură a codului, prin organizarea lor în module şi pachete.

În limbajul Python sunt predefinite mai multe funcţii care sunt des utilizate în programe, cum ar fi funcţii de conversie (`int`, `str`, `bool` etc.), funcţii matematice (`min`, `max`, `sum` etc.), funcţii pentru crearea unei colecţii (`list`, `dict`, `set` etc.) ş.a.m.d. O listă completă a tuturor funcţiilor predefinite din limbajul Python poate fi consultată aici: https://docs.python.org/3/library/functions.html.

O funcţie poate fi definită de către un utilizator folosind cuvântul cheie `def`, astfel:

```
def nume_funcție(parametrii formali)
      instrucțiuni
```

Parametrii formali sunt utilizaţi în antetul unei funcţii pentru a preciza datele sale de intrare într-un mod abstract (formal). *Parametrii efectivi* ai unei funcţii sunt expresii ale căror valori sunt asociate parametrilor formali în momentul apelării unei funcţii.

Parametrii formali ai unei funcţii pot fi de mai multe tipuri în limbajul Python:

- *parametri simpli* au, de obicei, un caracter poziţional, respectiv în momentul apelării unei funcţiei trebuie să fie înlocuiţi cu acelaşi număr de parametrii efectivi, compatibili ca tipuri de date:

```
def suma(x, y):
    return x + y


# apelare pozițională
s = suma(3, 7)
print("s =", s)

# apelare prin nume
s = suma(y=7, x=3)
print("s =", s)
```

  În acest exemplu, parametrii formali ai funcţiei `suma` sunt `x` şi `y`, iar constantele `3` şi `7` sunt parametrii efectivi. Deoarece în limbajul Python nu se precizează explicit tipurile

de date ale parametrilor formali, compatibilitatea dintre ei și parametrii efectivi poate fi, în unele cazuri, o noțiune destul de vagă. Astfel, în acest exemplu, parametrii efectivi ai funcției pot fi și două șiruri de caractere sau două liste, dar nu pot fi, de exemplu, un număr și un șir de caractere! De asemenea, se observă faptul că parametrii simpli permit și apelarea prin nume.

- *parametri cu valori implicite* sunt utilizați dacă se dorește inițializarea lor cu niște valori implicite, care vor fi utilizate în momentul apelării funcției cu un număr de parametrii efectivi mai mic decât numărul parametrilor formali:

```
def suma(x=0, y=0):
    return x + y


s = suma()              # x = 0 și y = 0
print("s =", s)         # s = 0

s = suma(7)             # x = 7 și y = 0
print("s =", s)         # s = 7

s = suma(y=19)          # x = 0 și y = 19
print("s =", s)         # s = 19

s = suma(7, 10)         # x = 7 și y = 10
print("s =", s)         # s = 17
```

Atenție, dacă antetul unei funcții conține și parametri simpli și parametri cu valori implicite, parametrii poziționali trebuie să fie precizați primii, pentru a evita ambiguitățile care pot să apară în momentul apelării funcției! De exemplu, dacă funcția se mai sus ar avea antetul `suma(x=0, y)`, atunci apelul `s = suma(7)` ar putea fi interpretat fie ca `s = suma(7, y)` și ar fi incorect (parametrului `y` nu i-ar fi asociat niciun parametru efectiv), fie ca `s = suma(0, 7)` și ar fi corect. În schimb, antetul `suma(x, y=0)` nu va mai genera nicio ambiguitate în cazul apelului `s = suma(7)`, putând fi interpretat doar ca `s = suma(7, 0)`.

În limbajul Python, se pot defini foarte simplu și funcții cu *număr variabil de parametri*, care sunt utile când nu se poate preciza numărul exact de parametrii ai unei funcții. De exemplu, în același program putem să avem nevoie de o funcție care să calculeze suma a două numere și de o funcție care să calculeze suma a trei numere. Deoarece în limbajul Python nu se pot defini două funcții cu același nume și număr diferit de parametri (de fapt, se pot defini, dar compilatorul o va lua în considerare doar pe ultima definită și orice apelare a primei funcții definite va fi semnalată ca eroare), înseamnă că ar trebui să definim două funcții cu nume diferite, dar care sunt foarte asemănătoare din punct de vedere al prelucrărilor efectuate.

Definirea unei funcții cu număr variabil de parametri se realizează adăugând simbolul * înaintea unui parametru, ceea ce indică faptul că parametrul respectiv va conține, sub forma unui tuplu, un număr oarecare de parametri efectivi. De exemplu, o funcție cu număr variabil de parametri care calculează suma acestora se poate defini și apela astfel:

```
def suma(*args):
    sa = 0
    for x in args:
        sa = sa + x
    return sa
s = suma()
print("s =", s)          #s = 0

s = suma(7)              #s = 7
print("s =", s)

s = suma(1, 2, 3, 4)     #s = 10
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista)
print("s =", s)          #s = 55
```

Evident, o funcție poate să aibă un singur parametru variabil, iar eventualii parametrii existenți după el trebuie să fie precizați explicit, prin numele lor, în momentul apelării funcției (deoarece ei nu mai pot fi accesați pozițional, fiind precedați de un număr necunoscut de parametrii). De exemplu, următoarea funcție calculează suma parametrilor variabili care sunt cel puțini egali cu valoarea `minim`:

```
def suma(*valori, minim):
    sv = 0
    for x in valori:
        if x >= minim:
            sv = sv + x
    return sv


s = suma(7, minim=5)             #s = 7
print("s =", s)

s = suma(1, 2, 3, minim=10)      #s = 0
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista, minim=8)
print("s =", s)                  #s = 27
```

Dacă în exemplul de mai sus nu vom preciza explicit valoarea parametrului `minim` (de exemplu, scriind `s = suma(1, 2, 3, 10)`), atunci se va semnala o eroare de tipul `TypeError` (pentru exemplul considerat aceasta va fi `TypeError: suma() missing 1 required keyword-only argument: 'minim'`).

Bineînțeles, dacă o funcție cu număr variabil de parametri are și parametri simpli și parametri cu valori implicite, atunci trebuie respectată și regula ca parametrii simpli să fie scriși înaintea celor cu valori implicite! De exemplu, următoarea funcție `suma` calculează suma parametrilor variabili cuprinși între valorile `minim` și `maxim` care sunt și multiplii ai numărului `t`:

```
def suma(t, *valori, minim=0, maxim=100):
    sv = 0
    for x in valori:
        if minim <= x <= maxim and x % t == 0:
            sv = sv + x
    return sv


s = suma(7, 14, 19, 21, -56)                          #s = 35
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100)              #s = -21
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100, maxim=0)     #s = -56
print("s =", s)
```

Datorită operației implicite de împachetare a mai multor valori sub forma unui tuplu, o funcție poate să furnizeze mai multe valori. Astfel, o instrucțiune de forma `return a, b` este echivalentă cu `return (a, b)`, așa cum se poate observa din următorul exemplu:

```
def suma_prod(x, y):
    return x + y, x * y


s, p = suma_prod(3, 7)
print("s =", s, "\tp =", p)

(s, p) = suma_prod(3, 7)
print("s =", s, "\tp =", p)

t = suma_prod(3, 7)
print("s =", t[0], "\tp =", t[1])
print("Suma si produsul:", *t)
```

Observați faptul că la apelarea funcției se poate utiliza operația de despachetare a unui tuplu, complementară celei de împachetare utilizată în instrucțiunea `return`!

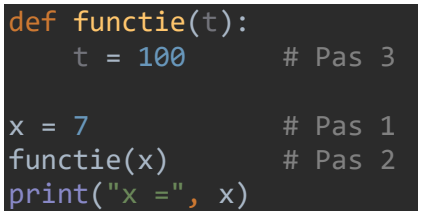
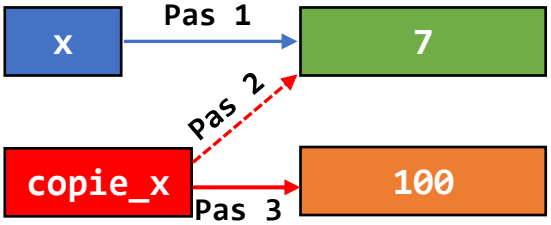
Modalităţi de transmitere a parametrilor

În limbajele C/C++ transmiterea unui parametru efectiv către o funcţie se poate realiza în două moduri:

- *transmitere prin valoare*: se transmite o copie a valorii parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcţiei NU se reflectă şi în exteriorul său;
- *transmitere prin adresă/referinţă*: se transmite adresa parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcţiei se reflectă şi în exteriorul său.

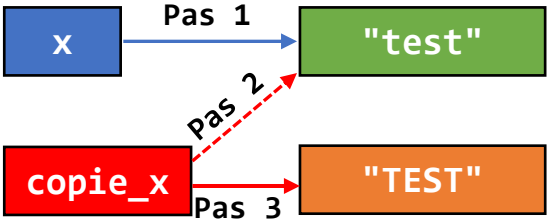
În limbajul Python, orice parametru efectiv al unei funcţii este o referinţă către un obiect (i.e., nu se transmite obiectul în sine, ci o referinţă spre el) care se transmite implicit prin valoare (i.e., se transmite o copie a referinţei respective), deci modificarea referinţei respective în interiorul funcţiei nu se va reflecta în exteriorul său. Din acest motiv, mecanismul de transmitere a parametrilor către o funcţie în limbajul Python se numeşte *transmitere prin referinţă la obiect* (*call by object reference*).

Exemplul 1:

Funcţia	Executarea programului	Rezultat
		

Explicaţie: După pasul 2, variabilele `x` şi `copie_x` vor conţine aceeaşi referinţă (spre obiectul 7), dar după pasul 3 doar `copie_x` se va modifica (va conţine referinţa obiectului 100), deoarece `x` este o referinţă transmisă prin valoare.

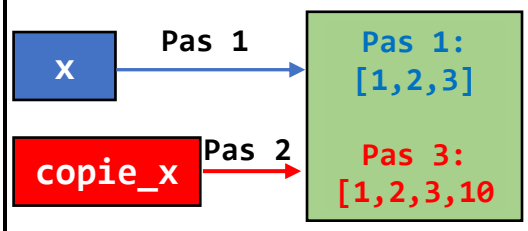
Exemplul 2:

Funcţia	Executarea programului	Rezultat
		

Explicaţie: vezi exemplul anterior!

Exemplul 3:

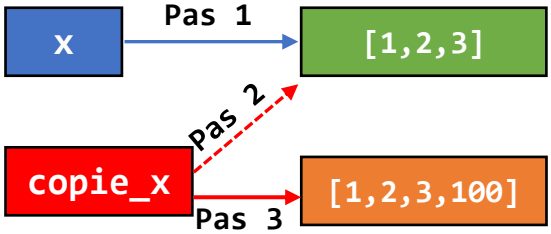
Funcția	Executarea programului	Rezultat

`def functie(t):`
`  t.append(100) # Pas 3`
 	
`x = [1, 2, 3]  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x =`
		`[1,2,3,100]`

Explicație: După pasul 2, variabilele x și `copie_x` vor conține aceeași referință, iar după pasul 3 conținutul listei se va modifica prin intermediul referinței din `copie_x` (însă fără a modifica referința listei!), deci variabila x va putea accesa lista modificată.

Exemplul 4:

Funcția	Executarea programului	Rezultat

`def functie(t):`
`  t = t + [100] # Pas 3`
 	
`x = [1, 2, 3]  # Pas 1`
`functie(x)  # Pas 2`
`print("x =", x)`
		`x =`
		`[1,2,3]`

Explicație: După pasul 2, variabilele x și `copie_x` vor conține aceeași referință (spre lista [1,2,3]), dar la pasul 3 operatorul de concatenare (+) va crea o nouă listă [1,2,3,100], deci după pasul 3 doar `copie_x` va conține referința noii liste. Practic, deși o listă este mutabilă, elementul 100 a fost adăugat într-o manieră specifică obiectelor imutabile!

În concluzie, mecanismul de *transmitere prin referință la obiect* a parametrilor efectivi către o funcție în limbajul Python utilizează transmiterea prin valoare a referințelor parametrilor efectivi (copii ale referințelor lor, deci modificarea acestora nu va fi vizibilă în exteriorul funcției) și acționează astfel:

- dacă obiectul asociat referinței este *imutabil*, atunci modificarea parametrului respectiv în interiorul funcției nu se va reflecta în exteriorul funcției deoarece conținutului obiectului asociat referinței nu poate fi modificat (din cauza imutabilității), iar crearea unei referințe noi nu se va reflecta în exteriorul funcției (din cauza transmiterii prin valoare a referinței);
- dacă obiectul asociat referinței este *mutabil*, atunci modificarea conținutului parametrului respectiv în interiorul funcției se va reflecta în exteriorul funcției deoarece nu se va modifica referința obiectului.

Atenție la exemplul 4 de mai sus, deoarece acolo nu se modifică direct conținutul obiectului mutabil de tip listă, ci se creează un nou obiect (tot o listă mutabilă) care conține noua listă și se atribuie referința sa copiei referinței parametrului!

Variabile locale și globale

O variabilă definită în interiorul unei funcţii se numeşte *variabilă locală* şi este vizibilă (i.e., poate fi utilizată) doar în interiorul funcţiei respective.

O variabilă definită în afara oricărei funcţii se numeşte *variabilă globală* şi este vizibilă în tot modulul respectiv (i.e., poate fi utilizată în interiorul oricărei funcţii).

Exemplu:

```
def afisare():
    print("x = ", x)      # se va utiliza variabila globală x,
                          # deoarece nu există o variabilă locală x
x = 100
afisare()                 # x = 100
```

Dacă într-o funcţie există definită o variabilă locală având acelaşi nume cu o variabilă globală, atunci, implicit, se va utiliza variabila locală în interiorul funcţiei:

```
def afisare():
    x = 200
    print("x = ", x)        # se va utiliza variabila locală x

x = 100
afisare()                   # x = 200
print("x = ", x)            # x = 100 (variabila globală)
```

Dacă într-o funcţie există definită o variabilă locală având acelaşi nume cu o variabilă globală, atunci putem utiliza variabila globală precizând în interiorul funcţiei acest lucru:

```
def afisare():
    global x
    print("x = ", x)        # se va utiliza variabila globală x
    x = 200

x = 100
afisare()                   # x = 100
print("x = ", x)            # x = 200
```

Atenţie, dacă în exemplul de mai sus ar lipsi declararea `global x`, atunci ar fi generată eroarea `UnboundLocalError: local variable 'x' referenced before assignment`, deoarece definirea unei variabile locale `x` prin `x=200` va determina interpretatorul să nu mai caute o variabilă globală cu numele `x`! Acelaşi lucru se va întâmpla şi în exemplul următor:

```
def f():
    x = x + 100             # eroare!
    print("x = ", x)
```

```
x = 200
f()
print("x = ", x)
```

În acest caz, eroarea apare deoarece interpretatorul consideră faptul că o instrucțiune de atribuire de forma "`x =`" reprezintă o declarare prin inițializare a unei variabile locale `x`, dar, evident, expresia `x + 100` nu poate fi evaluată, variabila locală `x` nefiind inițializată!

Funcții imbricate

În limbajul Python putem defini o funcție în interiorul altei funcții, așa cum se poate observa din exemplul următor:

```
def combinari(n, k):
    def factorial(x):
        p = 1
        for i in range(1, x+1):
            p = p * i
        return p

    return factorial(n) // (factorial(k) * factorial(n-k))

print(combinari(5, 3))
```

O funcție `g` definită în interiorul unei funcții `f` este locală funcției `f`, deci funcția `g` poate fi apelată doar în interiorul funcției `f` (i.e., funcția `g` este o funcție auxiliară pentru `f`).

O funcție imbricată are acces implicit la parametrii funcției în care este definită și la variabilele sale locale:

```
def calcul(x, y):
    n = 2
    def medie(k):
        return (x**n + y**n) / k

    return medie(2)

print("m = ", calcul(3, 4))              # m = 12.5
```

Atenție, deși o funcție imbricată are acces implicit la variabilele locale ale funcției în care este definită, ea nu le poate modifica deoarece va fi generată o eroare de tipul `UnboundLocalError`. Pentru a utiliza o variabilă locală într-o funcție imbricată, variabila locală trebuie declarată în interiorul funcției imbricate folosind cuvântul cheie `nonlocal`:

```
def f():
    n = 100
    def aux():
        nonlocal n
        n = n * 2

    aux()
    return n

print(f())              # 200
```

Practic, prin utilizarea cuvântului cheie `nonlocal` pentru declararea unei variabile în interiorul unei funcţii imbricate îi cerem interpretatorului să caute definirea variabilei respective în cel mai apropiat spaţiu de nume exterior funcţiei imbricate, mai puţin în cel global (pentru a accesa variabilele globale se foloseşte cuvântul cheie `global`, aşa cum deja am menţionat):

```
incr = 100
def f():
    n = 7
    incr = 10

    def g():
        nonlocal incr, n
        n = n + incr
        return n

    def h():
        global incr
        nonlocal n
        n = n + incr
        return n

    print("nonlocal incr:", g())    # nonlocal incr: 17
    print("global incr:", h())      # global incr: 117

f()
```

În limbajul Python, o funcţie poate să returneze o funcţie imbricată, aşa cum se poate observa din exemplul următor:

```
def putere(baza):


    def paux(exponent):
        return baza ** exponent

    return paux
```

```
putere10 = putere(10)

print(type(putere10))       # <class 'function'>
print(putere10.__name__)    # paux
print(putere10(3))          # 1000
```

Practic, `putere10` este o referinţă spre funcţia `paux` particularizată pentru `baza = 10`, deci poate fi utilizată la fel ca orice altă funcţie.

Transmiterea unei funcţii ca parametru al altei funcţii (callback)

Există mai multe situaţii în care este necesar să utilizăm *mecanismul de callback*, respectiv să transmitem o funcţie `f` ca parametru al unei funcţii `g`, astfel încât funcţia `g` să poată apela funcţia `f` când acest lucru este necesar. De exemplu, notificările utilizate în aplicaţiile mobile utilizează un mecanism asemănător mecanismului de callback, respectiv o aplicaţie de tip server înregistrează faptul că un anumit utilizator a acceptat să primească notificări şi în momentul în care apare un anumit eveniment pe server (de exemplu, când sunt depuşi sau retraşi bani dintr-un cont bancar), server-ul îi trimite utilizatorului respectiv o notificare. De asemenea, mecanismul de callback este intens utilizat în programarea interfeţelor grafice, respectiv sistemul de operare primeşte o funcţie pe care să o apeleze în momentul apariţiei unui anumit eveniment (e.g., apăsarea unui buton, închiderea unei ferestre, selectarea unei opţiuni dintr-o listă etc.).

Mecanismul de callback mai este utilizat şi în *programarea generică*, respectiv în scrierea unor funcţii care realizează prelucrări generice ale unei funcţii a cărei expresie nu este cunoscută (e.g., reprezentarea grafică a unei funcţii, calculul unei integrale etc.).

În continuare, vom prezenta o funcţie generică pentru calculul unor sume. De exemplu, să considerăm următoarele 3 sume:

$$S_1 = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$$
$$S_2 = 1^2 + 2^2 + \cdots + n^2$$
$$S_3 = e^1 + e^2 + \cdots + e^n$$

Evident, putem să definim câte o funcţie pentru calculul fiecărei sume, dar această soluţie nu este scalabilă. Putem observa cu uşurinţă faptul că toate cele 3 sume au următoarea formă generală:

$$S_k = \sum_{i=1}^{n} f_k(i)$$

unde prin $f_k(i)$ am notat termenul de rang i al sumei S_k pentru $k \in \{1, 2, 3\}$, deci, pentru sumele de mai sus, termenii generali sunt $f_1(i) = \frac{1}{i}$, $f_2(i) = i^2$ şi $f_3(i) = e^i$. Astfel, putem scrie o funcţie generică pentru calculul unei astfel de sume, care va avea parametrii `n` şi `fk`, unde `fk` va fi o funcţie care implementează termenul general al unei anumite sume:

```
import math

def suma_generica(n, fk):
    s = 0
    for i in range(1, n+1):
        s = s + fk(i)
    return s

def fk_1(i):
    return 1/i

def fk_2(i):
    return i**2


n = 10
s = suma_generica(n, fk_1)
print("Suma 1:", s)                 # Suma 1: 2.9289682539682538

s = suma_generica(n, fk_2)
print("Suma 2:", s)                 # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s)                 # Suma 3: 34843.77384533132
```

Observaţi faptul că putem apela funcţia `suma_generica` utilizând pentru termenul general şi funcţii predefinite!

Funcţii anonime (lambda expresii)

O *funcţie anonimă* (*lambda expresie*) este o funcţie foarte simplă, fără nume, definită folosind cuvântul cheie lambda, astfel:

```
lambda parametrii: expresie
```

O funcţie anonimă poate să aibă unul sau mai mulţi parametrii, dar corpul său trebuie să fie format dintr-o singură expresie (e.g., nu poate conţine instrucţiuni sau mai multe expresii). În momentul apelării unei funcţii anonime, expresia asociată va fi evaluată, iar rezultatul obţinut va fi furnizat direct, fără a fi necesară utilizarea cuvântului cheie `return` în interiorul funcţiei anonime.

Exemple:

- *suma a două numere:* `lambda x, y: x+y`
- *testarea parităţii unui număr întreg:* `lambda x: x % 2 == 0`
- *testarea divizibilităţii:* `lambda x, y: False if y == 0 else x % y == 0`
- *suma cifrelor unui număr:* `lambda x: sum([int(cf) for cf in str(x)])`

- *numărul vocalelor dintr-un şir:* `lambda sir: len([lit for lit in sir if lit in "aeiouAEIOU"])`
- *numărul valorilor dintr-o listă* `L` *strict mai mari decât o valoare* `v`*:* `lambda L, v: len([x for x in L if x > v])`

O funcţie anonimă poate fi apelată direct, în momentul definirii sale, lucru care nu este posibil în cazul funcţiilor obişnuite:

```
print((lambda x, y: x+y)(5, 7))      # 12
print((lambda s: len([c for c in s if c in "aeiou"]))("examene"))# 4
print((lambda x, y: x % y == 0)(24, 6))      # True
```

O referinţă spre o funcţie anonimă poate fi atribuită unei variabile, pentru a fi utilizată ca o funcţie obişnuită, dar acest lucru nu se recomandă deoarece contrazice ideea de funcţie anonimă (se preferă definirea unei funcţii obişnuite, cu nume):

```
f = lambda x, y: x+y
s = f(5, 7)
print(f"s = {s}")    # s = 12
```

O funcţie poate returna o funcţie anonimă, permiţând astfel crearea unor funcţii particularizate în raport de anumite criterii:

```
def selectorFunctie(tip):
    if tip == "suma":
        return lambda x, y: x + y
    elif tip == "diferenta":
        return lambda x, y: x - y
    elif tip == "produs":
        return lambda x, y: x * y
    else:
        return None


f = selectorFunctie("suma")
s = f(5, 7)
print(f"s = {s}")  # s = 12

f = selectorFunctie("produs")
p = f(5, 7)
print(f"p = {p}")  # p = 35
```

O funcţie anonimă poate fi utilizată în cadrul mecanismului de callback, pentru a elimina definiţiile funcţiilor transmise ca parametrii (evident, dacă funcţiile respective sunt suficient de simple pentru a fi implementata ca nişte funcţii anonime):

```
import math

def suma_generica(n, fk):
    s = 0
    for i in range(1, n+1):
        s = s + fk(i)
    return s


n = 10
s = suma_generica(n, lambda x: 1 / x)
print("Suma 1:", s)                    # Suma 1:
2.9289682539682538

s = suma_generica(n, lambda x: x ** 2)
print("Suma 2:", s)                    # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s)                    # Suma 3:
34843.77384533132
```

Încheiem prin a preciza faptul ca funcţiile anonime sunt utilizate, de obicei, în *programarea funcţională*, o altă paradigmă de programare implementată în limbajul Python, alături de programarea procedurală şi programarea orientată pe obiecte (https://realpython.com/python-functional-programming/).

Sortarea colecţiilor de date

În limbajul Python, colecţiile pot fi sortate crescător folosind funcţia predefinită `sorted`, care furnizează o listă cu elementele colecţiei respective sortate crescător:

- `sorted([2, 1, 5, 2, 1, 4]) = [1, 1, 2, 2, 4, 5]`
- `sorted((2, 1, 5, 2, 1, 4)) = [1, 1, 2, 2, 4, 5]`
- `sorted({2, 1, 5, 2, 1, 4}) = [1, 2, 4, 5]`
- `sorted({"d": 2, "a": 1, "f": 0, "b": 3 }) = ['a', 'b', 'd', 'f']`

Observaţi faptul că, indiferent de tipul colecţiei sortate (i.e., listă, tuplu sau mulţime), rezultatul este furnizat sub forma unei liste, iar în cazul unui dicţionar funcţia va furniza doar o listă a cheilor dicţionarului sortate crescător. Dacă dorim să obţinem perechile unui dicţionar sortate crescător în funcţie de chei, trebuie să sortăm o listă cu tuplurile corespunzătoare intrărilor sale:

```
sorted({"d": 2, "a": 1, "b": 3, "f": 0}.items()) = [('a', 1), ('b', 3),
('d', 2), ('f', 0)]
```

În cazul sortării unui şir de caractere, funcţia `sorted` va returna o listă formată din caracterele şirului, ordonate crescător:

```
sorted("exemplu") = ['e', 'e', 'l', 'm', 'p', 'u', 'x']
```

Dacă este necesar, putem să transformăm lista de caractere furnizată de metoda `sorted` înapoi într-un şir de caractere, folosind metoda `join`, astfel:

```
"".join(sorted("exemplu")) = "eelmpux"
```

Observaţie: O listă poate fi sortată direct, prin rearanjarea elementelor sale în ordine crescătoare, folosind metoda `sort` din clasa `list`:

```
L = [2, 1, 5, 2, 1, 4]
L.sort()
print("L =", L)             # L = [1, 1, 2, 2, 4, 5]
```

Dacă ulterior nu mai avem nevoie de lista iniţială, atunci se recomandă utilizarea metodei `sort` în locul funcţiei `sorted`, deoarece nu utilizează memorie suplimentară!

Deoarece funcţia `sorted` şi metoda `sort` din clasa `list` au aceiaşi parametrii opţionali, în continuare vom prezenta doar funcţia `sorted`, deoarece ea poate fi utilizată pentru orice structură de date iterabilă.

Pentru a realiza o sortare descrescătoare a elementelor unei structuri de date iterabile, parametrul opţional `reverse` al funcţiei `sorted` trebuie setat la valoarea `True`:

- `sorted([2, 1, 5, 2, 1, 4], reverse=True) = [5, 4, 2, 2, 1, 1]`
- `"".join(sorted("exemplu", reverse=True)) = "xupmlee"`

Evident, sortarea unei structuri de date iterabile se poate realiza doar în cazul în care elementele sale sunt comparabile, altfel se va genera o eroare de tipul `TypeError`. De exemplu, în cazul apelului `sorted([100, -10, "12345", 70])` se va genera eroarea `TypeError: '<' not supported between instances of 'str' and 'int'`!

Dacă o listă este formată din tupluri comparabile, atunci acestea vor fi sortate în ordine lexicografică. De exemplu, prin apelul `sorted([('g', 1), ('f', 7), ('b', 3), ('a', 2), ('f', 0), ('b', 1)])` se va obţine lista `[('a', 2), ('b', 1), ('b', 3), ('f', 0), ('f', 7), ('g', 1)]`, în care tuplurile au fost sortate în ordinea crescătoare a primelor componente, iar în cazul în care acestea erau egale în ordinea crescătoare a componentelor secundare. În cazul unei liste de şiruri de caractere, sortarea se va realiza folosind tot ordinea lexicografică. De exemplu, prin apelul `sorted(["prune", "mere", "ananas", "pere", "mango"])` se va obţine lista `["ananas", "mango", "mere", "pere", "prune"]`.

Pentru realizarea unor sortări complexe, eventual bazate pe mai multe criterii, putem să asociem fiecărui element al unei colecţii o *cheie* pe bază căreia să se realizeze operaţia de sortare. Acest lucru se realizează folosind parametrul opţional `key` al funcţiei `sorted`,

respectiv atribuindu-i acestuia numele unei funcţii care asociază unui element al colecţiei cheia dorită. De exemplu, pentru a sorta crescător numerele dintr-o listă în funcţie de sumele cifrelor lor, vom folosi pentru parametrul opţional `key` funcţia `sumaCifre`, care calculează suma cifrelor unui număr natural `nr`:

```
def sumaCifre(nr):
    return sum([int(c) for c in str(nr)])

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=sumaCifre))
```

În urma executării programului, se va afişa următoarea listă (am evidenţiat grupurile de numere care au aceeaşi sumă a cifrelor):

$$[\underbrace{101}_{2}, \underbrace{30, 111, 12}_{3}, \underbrace{202}_{4}, \underbrace{71, 107}_{8}, \underbrace{27, 81, 18}_{9}]$$

Practic, funcţia `sumaCifre` a fost apelată pentru fiecare element al listei înainte ca acesta să fie comparat cu alte elemente, iar valoarea obţinută (cheia elementului) a fost utilizată în comparări în locul numărului respectiv!

Deoarece funcţia `sorted` implementează *o metodă de sortare stabilă*, în lista sortată se va păstra ordinea relativă din lista iniţială a elementelor cu chei egale. De exemplu, în lista iniţială, numărul 27 se afla înaintea numărului 81, iar numărul 81 se afla înaintea numărului 18, iar această ordine relativă este păstrată şi în lista sortată.

Putem rescrie mai concis secvenţa de cod de mai sus, utilizând o funcţia anonimă în locul funcţiei `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: sum([int(c) for c in str(n)])))
```

Dacă dorim să sortăm numerele din listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm crescător după valorile lor, atunci vom asocia fiecărui număr un tuplu format din suma cifrelor sale şi el însuşi:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, nr

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]

print("L =", sorted(L, key=sumaCifre))
```

Astfel, se va afişa lista `[101, 12, 30, 111, 202, 71, 107, 18, 27, 81]`, deoarece cheile asociate elementelor listei iniţiale `[30, 27, 111, 71, 101, 107, 12, 81, 202, 18]` au fost tuplurile `(3, 30)`, `(9, 27)`, `(3, 111)`, `(8, 71)`, `(2, 101)`, `(8, 107)`, `(3, 12)`, `(9, 81)`, `(4, 202)`, `(9, 18)`, iar sortarea elementelor listei a fost realizată comparând lexicografic aceste tupluri!

Putem rescrie mai concis secvența de cod de mai sus, utilizând o funcție anonimă care furnizează un tuplu, în locul funcției `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: (sum([int(c) for c in str(n)]), n)))
```

Observație: Pentru a sorta descrescător o colecție în funcție de o cheie numerică k este suficient să folosim cheia -k. De exemplu, pentru a sorta numerele dintr-o listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm descrescător după valorile lor, atunci vom utiliza următoarea funcție pentru chei:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, -nr
```

Funcția utilizată pentru a calcula cheia unui element poate fi și o funcție predefinită. De exemplu, putem utiliza funcția predefinită `len` pentru a sorta o listă de șiruri de caractere în ordinea crescătoare a lungimilor lor:

```
L = ["prune", "pere", "ananas", "mere", "mango"]
print(sorted(L, key=len))   # ['pere', 'mere', 'prune', 'mango', 'ananas']
```

În continuare, vom mai prezenta câteva exemple de sortări complexe:

a) Să se sorteze o listă de numere naturale astfel încât numerele pare sortate crescător să fie poziționate înaintea celor impare sortate descrescător.

   Pentru a realiza această sortare, vom asocia fiecărui număr natural `nr` cheia `(0, nr)` dacă el este par, respectiv cheia `(1, -nr)` dacă el este impar. Practic, prima componentă a cheii este chiar restul împărțirii numărului `nr` la 2 (i.e., paritatea sa), iar cea de-a doua componentă este utilizată pentru a sorta crescător sau descrescător numerele cu aceeași paritate!

```
def paritate(nr):
    return nr % 2, nr if nr % 2 == 0 else -nr


L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=paritate)
print("Lista sortata:", L)
```

   De exemplu, dacă lista inițială este `[52, 27, 111, 71, 101, 17, 107, 12, 18]`, atunci, după sortare, se va obține lista `[12, 18, 52, 111, 107, 101, 71, 27, 17]`.

   Putem utiliza o funcție anonimă în locul funcției `paritate`:

```
L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=lambda n: (0, n) if n % 2 == 0 else (1, -n))
print("Lista sortata:", L)
```

b) Considerăm o listă care conține informații despre mai mulți studenți, respectiv pentru fiecare student se cunoaște numele, grupa și nota obținută la examenul de admitere. Să se sorteze studenții în ordinea crescătoare a grupelor, în fiecare grupă studenții să fie sortați descrescător după nota obținută la examenul de admitere, iar în cazul unor note egale studenții să fie sortați alfabetic.

Vom considera faptul că informațiile despre fiecare student sunt memorate într-un tuplu, astfel:

```
L = [( "Popescu Ion",  131, 9.25),
     ( "Ionescu Ana",  133, 8.75),
     ( "Popa Marian",  131, 9.85),
     ( "David Maria",  132, 8.95),
     ("Gheorghe Ana",  131, 9.85),
     ("Popescu Anca",  132, 9.15),
     ("Corbu Florin",  133, 8.05),
     ("Gheorghe Dan",  132, 9.15)]
```

Cheia asociată unui student va consta din componentele tuplului respectiv, în ordinea specificată în criteriile de sortare:

```
def cheie_student(t):
    return t[1], -t[2], t[0]

S = sorted(L, key=cheie_student)
print(*S, sep="\n")
```

După sortarea listei, studenții vor fi afișați în următoarea ordine:

```
("Gheorghe Ana",  131, 9.85)
( "Popa Marian",  131, 9.85)
( "Popescu Ion",  131, 9.25)
("Gheorghe Dan",  132, 9.15)
("Popescu Anca",  132, 9.15)
( "David Maria",  132, 8.95)
( "Ionescu Ana",  133, 8.75)
("Corbu Florin",  133, 8.05)
```

Putem simplifica secvența de cod utilizând o funcție anonimă în locul funcției `cheie_student`:

```
S = sorted(L, key=lambda t: (t[1], -t[2], t[0]))
print(*S, sep="\n")
```

c) Să se sorteze șirurile de caractere dintr-o listă `L` în ordinea descrescătoare a lungimilor prefixelor maximale comune cu un șir dat `s`. De exemplu, dacă lista este `L = ["apasator", "apartament", "exemplu", "ars", "test", "aparat", "amic"]` și `s = "aparator"`, atunci lista sortată va fi `L = ["aparat", "apartament", "apasator", "ars", "amic", "exemplu", "test"]`.

Pentru a determina prefixul maximal comun a două şiruri, vom considera, pe rând, toate prefixele unuia dintre şiruri, în ordinea descrescătoare a lungimilor, şi vom verifica dacă el este prefix şi pentru cel de-al doilea şir:

```
def prefixMaxim(s, t):
    for i in range(len(s), 0, -1):
        if t.startswith(s[:i]):
            return i
    return 0
```

Evident, funcţia se poate optimiza din punct de vedere al complexităţii computaţionale considerând prefixele celui mai scurt şir dintre cele două!

Din păcate, funcţia `prefixMaxim` are 2 parametri, deci nu o putem utiliza pentru a furniza cheile asociate şirurilor din listă (funcţia ar trebui să aibă un singur parametru, respectiv un element al listei)! Totuşi, putem să rezolvăm această problemă folosind funcţii imbricate, respectiv definim funcţia `prefixMaxim` cu un singur parametru, şirul dat `s`, şi în interiorul său definim o funcţie auxiliară `pmax` cu un singur parametru `t`, care va determina prefixul maximal comun dintre şirurile `s` şi `t`, iar funcţia `prefixMaxim` va returna funcţia `pmax`:

```
def prefixMaxim(s):
    def pmax(t):
        for i in range(len(s), 0, -1):
            if t.startswith(s[:i]):
                return i, t
        return 0, t

    return pmax
```

Astfel, prin apelul `prefixMaxim(s)`, vom obţine o funcţie cu un singur parametru, care va determina lungimea prefixului comun maximal dintre un şir oarecare `t` (parametrul funcţiei) şi şirul dat `s`, iar această funcţie poate fi utilizată pentru a furniza cheia asociată unui element:

```
L = ["apasator", "apartament", "exemplu", "ars", "test",
     "aparat", "amic"]
s = "aparator"

L.sort(key=prefixMaxim(s), reverse=True)
print(L)
```

Cum ar trebui să modificăm funcţia `prefixMaxim` astfel încât şirurile având aceeaşi lungime a prefixului comun maximal să fie sortate alfabetic? În acest exemplu, nu putem înlocui funcţia `prefixMaxim` cu o funcţie anonimă. De ce?

În încheiere, menţionăm faptul că funcţia `sorted` şi metoda `sort` implementează algoritmul de sortare *Timsort*, care a fost creat special în 2002 de Tim Peters pentru a fi

implementat în limbajul Python (https://en.wikipedia.org/wiki/Timsort). Algoritmul *Timsort* este derivat din algoritmii de sortare prin interclasare (*Mergesort*) și sortare prin inserție (*Insertion sort*), având complexitatea $\mathcal{O}(n\log_2 n)$ în cazul cel mai defavorabil.

Funcții recursive

În general, prin *recursivitate* se înțelege proprietatea unor noțiuni de a se defini prin ele însele. De exemplu, numerele naturale poate fi definite recursiv folosind următoarele două axiome ale lui Peano (https://en.wikipedia.org/wiki/Peano_axioms): "*Zero este un număr natural.*" și "*Succesorul oricărui număr natural este tot un număr natural.*".

În programare, o *funcție recursivă* este o funcție care se autoapelează, direct sau indirect.

Deoarece recursivitatea indirectă (i.e., o funcție `f` apelează o funcție `g`, iar funcția `g` apelează, la rândul său, funcția `f`) este foarte rar utilizată în programare (de exemplu, pentru a calcula media aritmetico-geometrică: https://en.wikipedia.org/wiki/Arithmetic-geometric_mean), în continuare vom prezenta doar recursivitatea directă.

Un exemplu clasic de funcție recursivă îl reprezintă calculul factorialului unui număr natural n (i.e., $n! = 1 \cdot 2 \cdot \ldots \cdot n$), folosind următoarea relație de recurență:

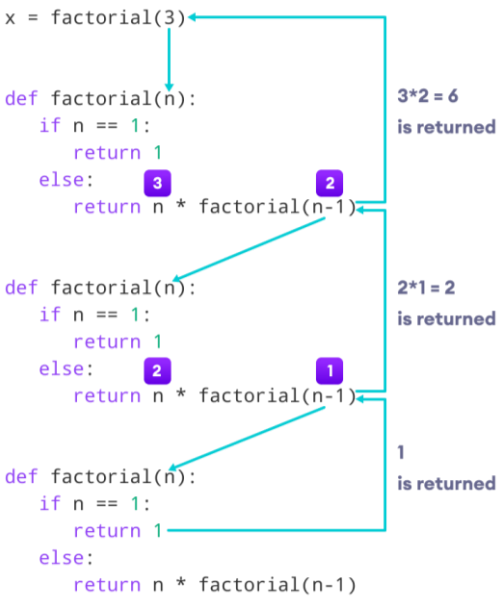
$$n! = \begin{cases} 1, & \text{dacă } n = 1 \\ n \cdot (n-1)!, & \text{dacă } n \geq 1 \end{cases}$$

O funcție care implementează în limbajul Python relația de recurență de mai sus este următoarea:

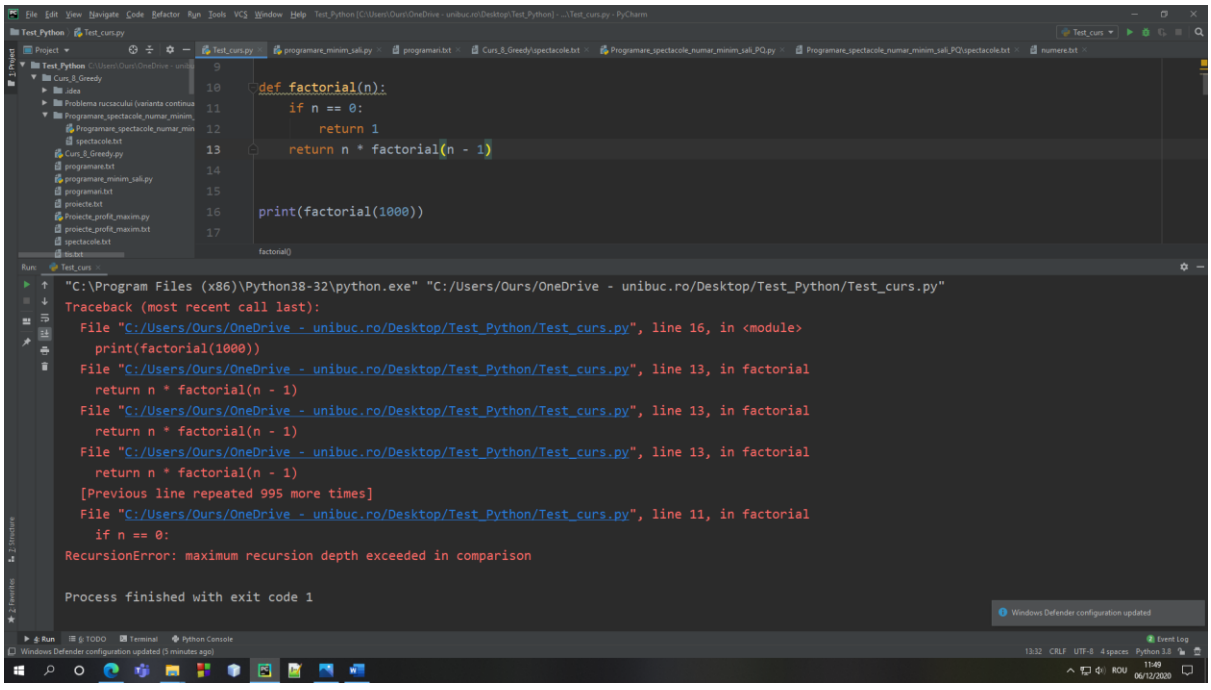
```python
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

În momentul apelării unei funcții, *contextul de apel* asociat (numele, variabilele locale, parametrii, adresa de revenire etc.) se salvează în stiva alocată programului, iar în momentul terminării executării sale, contextul de apel este eliminat din stivă.

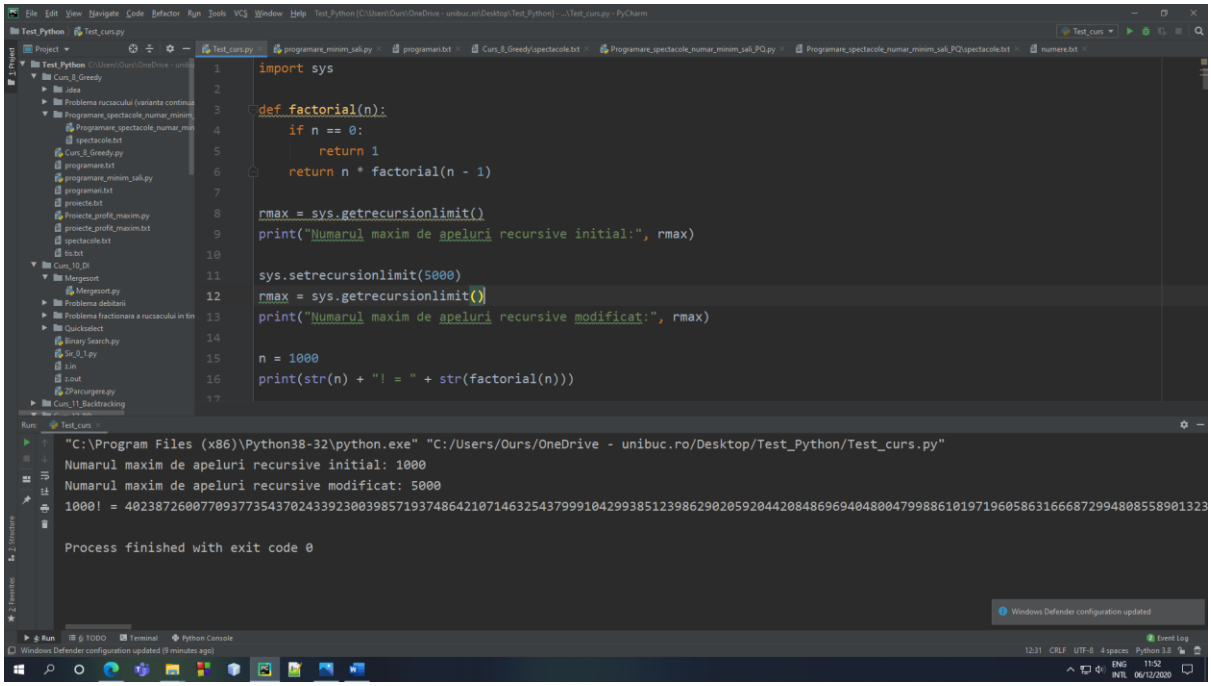
De exemplu, pentru apelul `f = factorial(3)`, stiva programului (reprezentată invers) va avea următoarea evoluție (sursa imaginii: https://www.programiz.com/python-programming/recursion):



Observaţie: Orice funcţie recursivă trebuie să conţină, pe lângă componenta recurentă (care conţine autoapelurile), şi o condiţie de oprire (care nu conţine niciun autoapel) realizabilă. În caz contrar, în momentul apelării sale, se va produce o recursivitate "infinită", care va conduce la depăşirea numărului maxim de apeluri recursive permis (implicit, acesta este egal cu 1000) şi la apariţia unei erori. De exemplu, în cazul apelului `f = factorial(1000)`, se va afişa următoarea eroare:



Numărul maxim de apeluri recursive care pot fi salvate pe stivă poate fi modificat astfel:



În continuare, vom prezenta câteva exemple clasice de funcţii recursive:

a) *Şirul lui Fibonacci* este definit prin următoarea relaţie de recurenţă:

$$f_n = \begin{cases} 0, & \text{dacă } n = 0 \\ 1, & \text{dacă } n = 1 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 2 \end{cases}$$

O implementare directă a acestei relaţii, care va furniza valoarea termenului de rang n a şirului lui Fibonacci, este următoarea funcţie recursivă:

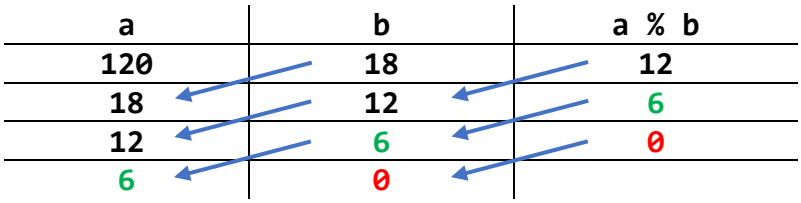
```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-2) + fibo(n-1)
```

Pentru $n \geq 40$ se observă faptul că timpul de executare este destul de mare şi creşte în raport cu valoarea lui n. În capitolul următor, dedicat complexităţii computaţionale, se va demonstra faptul că numărul de apeluri recursive efectuate este exponenţial în raport cu n, aceasta fiind cauza timpului mare de executare. O implementare eficientă se poate obţine iterativ:

```
def fibo(n):
    if n <= 1:
        return n

    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a
```

b) *Algoritmul lui Euclid* permite determinarea celui mai mare divizor comun a două numere întregi nenule a şi b prin împărţiri repetate, respectiv: cât timp restul împărţirii lui a la b este nenul înlocuim a cu b şi b cu restul împărţirii lui a la b, iar ultimul rest nenul obţinut va fi $cmmdc(a, b)$. De exemplu, pentru $a = 120$ şi $b = 18$, cel mai mare divizor comun se va calcula astfel:



Ultimul rest nenul este egal cu 6, deci vom obţine $cmmdc(120, 18) = 6$. Se observă faptul că ultimul rest nenul este egal cu ultimul împărţitor, deci o variantă de implementare iterativă a acestui algoritm este următoarea:

```
def cmmdc(a, b):
    r = a % b
    while r != 0:
        a, b = b, r
        r = a % b
    return b
```

Analizând algoritmul, putem deduce foarte uşor următoarea formulă de recurenţa pentru calculul celui mai mare divizor comun a două numere întregi:

$$cmmdc(a, b) = \begin{cases} a, & \text{dacă } b = 0 \\ cmmdc(b, a\%b), & \text{dacă } b \neq 0 \end{cases}$$

Implementarea acestei relaţii de recurenţă printr-o funcţie recursivă este foarte simplă:

```
def cmmdc(a, b):
    if b == 0:
        return a
    return cmmdc(b, a%b)
```

Observăm faptul că funcţiile au complexităţi egale, respectiv numărul de iteraţii este aproximativ egal cu numărul de autoapeluri.

c) *Calculul sumei cifrelor unui număr natural* se poate realiza folosind următoarea relaţie de recurenţă:

$$sc(n) = \begin{cases} n, & \text{dacă } n < 10 \\ n\%10 + sc(n//10), & \text{dacă } n \geq 10 \end{cases}$$

Implementarea acestei relaţii de recurenţă printr-o funcţie recursivă este banală:

```
def sc(n):
    if n < 10:
        return n
    return n%10 + sc(n//10)
```

Ce relație există între numărul de autoapeluri din varianta recursivă și numărul de iterații din varianta iterativă?

d) *Suma elementelor dintr-o listă* se poate defini recursiv ca fiind suma dintre primul element al listei și suma elementelor din restul listei dacă lista este nevidă, respectiv 0 dacă lista este vidă:

```
def suma_lista(L):
    if len(L) == 0:
        return 0
    return L[0] + suma_lista(L[1:])
```

Într-un mod similar se poate calcula și suma elementelor strict pozitive dintr-o listă:

```
def sumapoz_lista(L):
    if len(L) == 0:
        return 0
    if L[0] > 0:
        return L[0] + sumapoz_lista(L[1:])
    else:
        return sumapoz_lista(L[1:])
```

e) *Frecvența unei litere într-un șir de caractere* se poate calcula recursiv astfel:

```
def frecventa(litera, sir):
    if len(sir) == 0:
        return 0
    return int(sir[0] == litera) + frecventa(litera, sir[1:])
```

f) Din numărul 4 se poate obține orice număr natural nenul aplicând în mod repetat următoarele operații:
1. împărțirea numărului la 2;
2. adăugarea cifrei 4 la sfârșitul numărului;
3. adăugarea cifrei 0 la sfârșitul numărului.

De exemplu, numărul 101 se poate obține prin șirul de operații $4 \to 40 \to 404 \to 202 \to 101$, iar numărul 133 prin șirul de operații $4 \to 2 \to 24 \to 12 \to 6 \to 64 \to 32 \to 16 \to 8 \to 84 \to 42 \to 424 \to 212 \to 106 \to 1064 \to 532 \to 266 \to 133$.

Pentru a afișa șirul de operații prin care se poate obține un număr natural nenul din numărul 4, vom aplica asupra sa operațiile inverse operațiilor date:
1'. înmulțirea numărului cu 2;
2'. eliminarea ultimei cifre, dacă aceasta este 4;
3'. eliminarea ultimei cifre, dacă aceasta este 0.

De exemplu, pentru numărul 101 vom obține următorul șir de operații 101 → 202 → 404 → 40 → 4.

```
def numar4(n):
    if n != 4:
        if n % 10 == 0 or n%10 == 4:
            numar4(n//10)
        else:
            numar4(2*n)
        print(" ->", n, end="")
    else:
        print(4, end="")
```

Încheiem prezentarea funcțiilor recursive menționând faptul că, în general, acestea consumă multă memorie (pentru salvarea contextelor de apel), sunt mai greu de depanat decât funcțiile iterative și, de multe ori, necesită un timp de executare mai mare. Din aceste motive, se recomandă utilizarea unei funcții recursive doar în cazul în care complexitatea sa computațională este echivalentă cu cea a variantei iterative, dar implementarea sa este mai simplă.

Generatori

Un *generator* este un tip de funcție a cărei executare nu se termină în momentul în care returnează o valoare, ceea ce îi permite furnizarea mai multor valori într-o manieră secvențială (i.e., sub forma unui *iterator*). Un exemplu de generator predefinit în limbajul Python este `range`, care furnizează valorile întregi dintr-un anumit interval una câte una. Pentru a returna valori în mod repetat, un generator folosește instrucțiunea `yield` în locul instrucțiunii `return`:

```
def generator_numere_pare(n):
    x = 0
    while x <= n:
        yield x
        x += 2
```

Valorile furnizate de un generator pot fi accesate secvențial, folosind, de exemplu, o instrucțiune `for`:

```
for x in generator_numere_pare(100):
    print(x, end=" ")
```

O altă posibilitate de accesare secvențială a valorilor furnizate de un generator o reprezintă utilizarea funcției `next(iterator, [valoare_implicită])`, care permite, în general, accesarea următoarei valori dintr-un iterator sau furnizarea unei valori implicite, precizată prin parametrul opțional `valoare_implicită`, în momentul în care iteratorul nu mai conține nicio valoare:

```
nr_pare = generator_numere_pare(100)
x = next(nr_pare, -1)
while x != -1:
    print(x, end=" ")
    x = next(nr_pare, -1)
```

Practic, pentru a putea furniza mai multe valori într-o manieră secvențială, contextul de apel al unui generator nu este eliminat de pe stiva programului în momentul executării unei instrucțiuni `yield`. Astfel, după revenirea dintr-un apel, un generator îți poate continua executarea din starea în care se afla înaintea apelului respectiv!

Instrucțiunea `return` vidă poate fi utilizată pentru a întrerupe executarea unui generator:

```
def generator_numere_pare(n):
    x = 0
    while True:
        yield x
        if x == n:
            return
        x += 2
```

Un generator poate rula "la infinit", așa cum se poate observa din următorul exemplu:

```
def generator_numere_pare():
    x = 0
    while True:
        yield x
        x = x + 2
```

Evident, în acest caz nu se pot utiliza modalitățile de accesarea a valorilor furnizate menționate anterior, ci accesarea acestora trebuie să fie întreruptă explicit:

```
for x in generator_numere_pare():
    print(x, end=" ")
    if x == 100:
        break
```

```
nr_pare = generator_numere_pare()
x = next(nr_pare, -1)
while x <= 100:
    print(x, end=" ")
    x = next(nr_pare, -1)
```

Întreruperea accesării valorilor furnizate de un generator infinit nu va conduce la oprirea sa, așa cum se poate observa din următorul exemplu (se vor afișa numerele pare de la 0 la 20):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
    if x == 10:
        break
```

```
for x in nr_pare:
    print(x, end=" ")
    if x == 20:
        break
```

Pentru a întrerupe forţat executarea unui generator, infinit sau nu, trebuie apelată metoda `close` (se vor afișa doar numerele pare de la 0 la 10):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
    if x == 10:
        break

nr_pare.close()

for x in nr_pare:
    print(x, end=" ")
    if x == 20:
        break
```

În concluzie, un generator reprezintă o modalitate foarte simplă de creare a unui iterator în limbajul Python. În plus, generatorii permit o utilizare eficientă a memoriei, valorile fiind furnizate secvenţial.